FPGASolve

Technical Report

Group SD0917

Matthew Nitschke

Richard Schultz

Swati Gupta

May 7, 2010

Table of Contents

Introduction:
The FPGASolve design project was created to obtain the Newton-Raphson power flow solution. Current power flow calculation software takes several seconds to obtain a solution and that time exponentially increases as the number of busses in the power system increases. The Newton-Raphson solution is an iterative method of solving the power flow, to solve the vector equation Ax=b the solver must run through several steps to solve the power flow. The FPGASolve projects uses an FPGA to solve the vector equation, by using hardware the power flow solution can be done faster than by using pure software. The FPGASolve project also uses the power flow solution to optimize a radial power distribution grid. The power flow is used to calculate the losses in the distribution system and calculates what configuration of the system will create the least amount of power loss. In order to do this the power flow solution must be run several times to create the optimal distribution layout. The FPGASolve project used a Altera DE2 development board with a Cyclone II FPGA.

Requirements:
- Identify the bottleneck of the current power flow analysis methods
- Hardware and software will be able to communicate with each other concurrently
- Software will run in-depth Newton-Raphson method solution
- Hardware will take inputs from software to complete Newton-Raphson method computations
- To improve runtime dramatically over pure software based computations
- Optimize radial distribution system using power solution.

Technical Content

Newton-Raphson Overview

The Newton Raphson method is a popular method of solving a system of simultaneous nonlinear equations. It uses the theory of the Taylor series expansion and an initial estimate to successively approximate closer and closer values of the answer. After a certain number of iterations, the approximation will converge with the real answer. To use it to solve for the system of nonlinear equations involved in a power system, we first had to calculate the bus admittance matrix.

Necessary input data

Before running the MATLAB code to obtain a power flow solution or optimize a network, we first had to obtain the necessary data of the distribution system. This data is divided into two categories: bus data and line data.
For the bus data, we needed to know the following for each bus in the system:

1. Bus Number: This is simply an arbitrary number assigned to each bus that is used to configure the system. Each bus must have a unique bus number.
2. Bus Code: This is a code used by the algorithm to determine what kind of bus it is. If it is a load bus, the code is a 0. Load buses require an initial voltage magnitude and angle to be set as well as the load MW and Mvar. If it is the slack

bus (feeder), the code is a 1. The only required information for the slack bus is the voltage magnitude and the phase angle. For voltage controlled buses, the code is a 2. For a voltage controlled bus, the voltage magnitude, generated MW, maximum limit of Mvar generation, and minimum limit of Mvar generation must all be set.

3. Voltage Magnitude: This is the per unit voltage magnitude of the bus.
4. Voltage Angle: This is phase angle of the voltage at the bus, entered in degrees.
5. Load MW: This is the load power at the bus. For the slack bus, this entry is zero.
6. Load Mvar: This is the load reactance at the bus. For the slack bus, this entry is zero.
7. Generated MW: This is the amount of real power generated at the bus.
8. Generated Mvar: This is the amount of reactance generated at the bus.
9. Minimum Generated Mvar: This is the lower limit of allowed Mvar generation at the bus. The entry for Generated Mvar cannot be lower than this, and the algorithm will not allow a configuration that would exceed this limit.
10. Maximum Generated Mvar: This is the upper limit of allowed Mvar generation at the bus. The entry for Generated Mvar cannot be higher than this, and the algorithm will not allow a configuration that would exceed this limit.
11. Injected Mvar: This is the amount of Mvars injected into the system via shunt capacitors at this bus.

We then had to place this data into a text file called busdata.txt that our MATLAB code would be able to access and read. We chose a simple format that is similar to the one used by Hadi Saadat in his textbook, "Power System Analysis, 2nd Edition". While some of our data is different, we chose to copy the format as it is very easy to use and easy to see what is going on. As long each piece of data is separated by a space, tab, or newline character (and in the correct order), MATLAB will read it in and put it where it needs to go in the busdata matrix. We used spaces to separate each entry for a bus, and new lines to separate the individual buses. An example can be seen below

	Bus Number	Bus Code	Voltage Magnitude	Voltage Angle	Load MW	Load Mvar	Generated MW	Generated Mvar	Min Mvar Generated	Max Mvar Generated	Injected Mvar
Bus 1	1	1	1.06	0	0	0	0	0	0	0	0
Bus 2	2	2	1.043	0	31.7	12.7	40	0	-40	50	0
Bus 3	3	0	1	0	12.4	11.2	0	0	0	0	0
Bus 4	4	0	1.06	0	7.6	6.6	0	0	0	0	15
Bus 5	5	2	1.01	0	4.2	2	0	0	-40	40	0

For the line data, we needed to know the following for each bus in the system:

1. Connected Bus 1: This is the bus number (from the bus data) of the first bus connected to the line.
2. Connected Bus 2: This is the bus number (from the bus data) of the second bus connected to the line.
3. Line Resistance: This is the per unit resistance of the line.
4. Line Reactance: This is the per unit reactance of the line.
5. Line Susceptance: This is one half of the per unit line charging susceptance.
6. Tap Setting: This is the tap setting of the transformer connected to the line. If the line is not connected to a transformer with an off-nominal tap setting, this value is just entered as 1.
7. Line Number: This is simply an arbitrary number assigned to the line that is used to configure the system. Each line must have a unique line number.
8. Breaker Code: This is a code used by the algorithm to determine whether the line is currently connected or not. If the line is connected, we used a 1 to denote that a line was connected, and a 0 to denote that it was not connected.

Once we had this data we entered it into a text file called breakerdata.txt that could be accessed and read by MATLAB. The formatting is the same as the one we used for the bus data, just with different values. An example of it can be seen below. It is important to note that it does not matter what order the Connected Bus 1 and Connected Bus 2 values are entered. While running the algorithm, both entries will be checked and the correct system will be configured. If there is a tap setting, however, Connected Bus 1 is assumed to be the on the tap side of the transformer.

	Connected Bus 1	Connected Bus 2	Line Resistance	Line Reactance	Line Susceptance	Tap Setting	Line Number	Breaker Code
Line 1	1	2	0.0192	0.0575	0.0164	1	1	1
Line 2	1	3	0.0452	0.1852	0.0204	1	2	1
Line 3	1	4	0.057	0.1737	0.0284	1	3	1
Line 4	2	3	0.0132	0.0379	0.0042	1	4	0
Line 5	2	5	0.0472	0.1983	0.0109	1	5	1

Equations:

The admittance of a line is defined as the inverse of the impedence. Since we know both the resistance and the reactance of each line from our input data, we can calculate our admittance (1/r+jx). From here, our bus admittance matrix is easy to find. The diagonal values are equal to the sum of the admittances that are connected to the bus that equate with the matrix row and column the element is in. For example, the entry for Y_{22} would be the sum of

the admittances of each line connected to bus 2. The off diagonal values are equal to the opposite of the admittance between 2 buses that are denoted by the matrix row and column of the element. For example Y_{34} would be equal to the opposite of the admittance between buses 3 and 4.

Once the bus admittance matrix has been formulated, we use it along with our voltage magnitude and phase angle information from our bus data to calculate the Jacobian. To do this, we use a set of relatively complicated equations, shown below.

The diagonal and the off-diagonal elements of $\boldsymbol{J_1}$ are

$$\frac{\partial P_i}{\partial \delta_i} = \sum_{j \neq i} |V_i||V_j||Y_{ij}| \sin(\theta_{ij} - \delta_i + \delta_j)$$

$$\frac{\partial P_i}{\partial \delta_j} = -|V_i||V_j||Y_{ij}| \sin(\theta_{ij} - \delta_i + \delta_j) \qquad j \neq i$$

The diagonal and the off-diagonal elements of $\boldsymbol{J_2}$ are

$$\frac{\partial P_i}{\partial |V_i|} = 2|V_i||Y_{ii}| \cos\theta_{ii} + \sum_{j \neq i} |V_j||Y_{ij}| \cos(\theta_{ij} - \delta_i + \delta_j)$$

$$\frac{\partial P_i}{\partial |V_j|} = |V_i||Y_{ij}| \cos(\theta_{ij} - \delta_i + \delta_j) \qquad j \neq i$$

The diagonal and the off-diagonal elements of $\boldsymbol{J_3}$ are

$$\frac{\partial Q_i}{\partial \delta_i} = \sum_{j \neq i} |V_i||V_j||Y_{ij}| \cos(\theta_{ij} - \delta_i + \delta_j)$$

$$\frac{\partial Q_i}{\partial \delta_j} = -|V_i||V_j||Y_{ij}| \cos(\theta_{ij} - \delta_i + \delta_j) \qquad j \neq i$$

The diagonal and the off-diagonal elements of $\boldsymbol{J_4}$ are

$$\frac{\partial Q_i}{\partial |V_i|} = -2|V_i||Y_{ii}| \sin\theta_{ii} - \sum_{j \neq i} |V_j||Y_{ij}| \sin(\theta_{ij} - \delta_i + \delta_j)$$

$$\frac{\partial Q_i}{\partial |V_j|} = -|V_i||Y_{ij}| \sin(\theta_{ij} - \delta_i + \delta_j) \qquad j \neq i$$

All of the information we have gathered is then put into the following format:

$$\begin{bmatrix} \boldsymbol{\Delta P} \\ \boldsymbol{\Delta Q} \end{bmatrix} = \begin{bmatrix} \boldsymbol{J_1} & \boldsymbol{J_2} \\ \boldsymbol{J_3} & \boldsymbol{J_4} \end{bmatrix} \begin{bmatrix} \boldsymbol{\Delta \delta} \\ \boldsymbol{\Delta |V|} \end{bmatrix}$$

Solving for this equation will give use a new values of $\Delta\delta$ and $\Delta|V|$, which we add to our initial estimates (usually 1 for a magnitude and 0 for a phase angle). This gives us new values of our voltage magnitudes and phase angles, which we use to find our new J, P and Q elements. Then we solve the system again. Each time we do this is called an iteration, and each iteration will bring us closer to the correct answer, until we are sufficiently close.

Newton-Raphson as performed on MATLAB

The purpose of our MATLAB code is to act as a top level algorithm. It will take in the necessary input data and set up the system of nonlinear equations necessary to solve the power flow solution in the proper format for the FPGA to use. Once the FPGA has solved this system of nonlinear equations and calculated the line flows and losses, the MATLAB then takes that data and uses it to find the optimal configuration for the radial power distribution system.

This is achieved through the use of several different subroutines, some of which are run many times throughout the course of the system's analysis. The first is called getdata.m, and is simply used to input the necessary system bus and breaker data by reading it out of a text file and then formatting it in such a way that it is easy to understand both for MATLAB and for the FPGA.

The next subroutine is called linedatas.m. It is a small subroutine that takes the breaker data from the getdata.m subroutine and extracts the appropriate line data from it.

Getybus.m is the next subroutine in the algorithm, and it takes the line data that was found in the linedatas.m subroutine and uses it to create the bus admittance matrix that is necessary to find the Jacobian, which is needed to set up the system of nonlinear equations that must be solved in order to calculate the line flows of the distribution system.

The next subroutine is newtonraphson.m. It is by far the largest of the MATLAB subroutines we had to write, and it contains both the formation of the Jacobian Matrix and the iterative solution of the power flow, but only when the software method is being used. When the FPGA is being used to solve set of nonlinear equations, the newtonraphson.m subroutine simply sets up the initial Jacobian and prepares it to be sent to the FPGA.

If the FPGA is being used, then the majority of the elements in the Jacobian matrix will be zeros. Instead of wasting time sending all of these zeros, we have written another MATLAB subroutine called CSRconv.m to convert the Jacobian found in newtonraphson.m into a compressed sparse row matrix. This is set of arrays has far fewer elements than the full Jacobian, and so is faster and easier to send to the FPGA.

In order to send this information to the FPGA, we have written a subroutine called serial.m, which controls the computer's RS-232 serial port (which is connected to the FPGA development board). First it opens the port, then sends the compressed sparse row matrix containing our Jacobian data. While the FPGA is calculating the solution, the serial.m subroutine waits to receive the information. After the FPGA sends that data back, the serial port is closed, and the algorithm moves on.

Once the system of nonlinear equations has been solved, whether it is by MATLAB or the FPGA, the result will then be used by the MATLAB subroutine getlineflow.m in order to calculate the line flows and losses. Using this information, we can then compare with other configurations to find the optimal one.

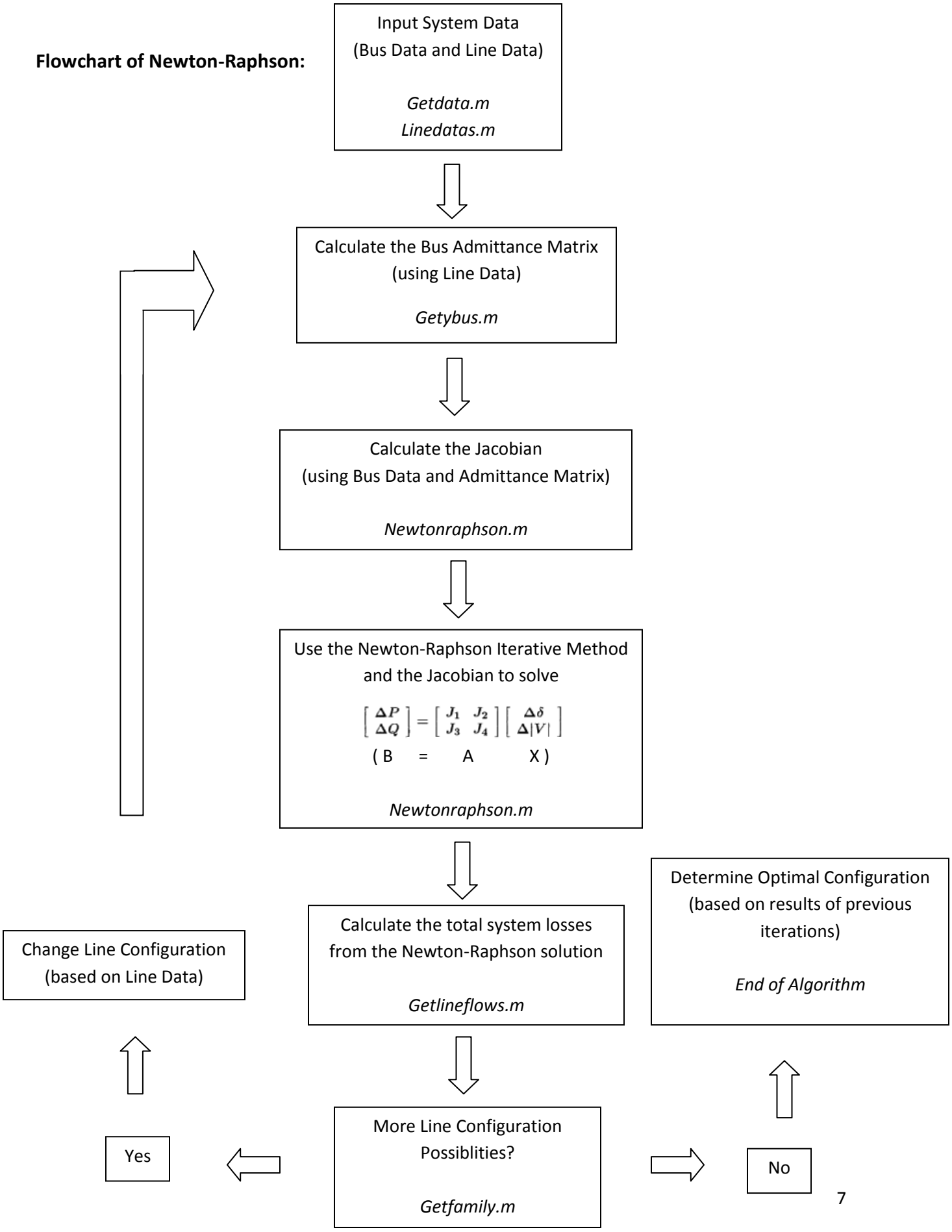
Finding the optimized configuration is actually what is done by the overall top level algorithm. Running through all of the subroutines explained above with different configuration data, it compares the different possible system configuration's line flows and losses to find the best way to configure our radial distribution system. In order to figure out which configurations should be tested, we wrote a subroutine called getfamily.m. This subroutine uses the initial breaker data to figure out the different possible lines that can be turned on and off while still keeping the system radial. After each new configuration is computed, all of the previous

subroutines run to calculate the line flows and losses. Once each possible configuration has been tested, the one with the lowest amount of losses is chosen to be output as the optimized system.

We actually have two separate algorithms that can be run. They are exactly the same as far as how they are computed, but their outputs are different. In one, the time that it takes the algorithm to complete is calculated and output at the end, along with the losses. In the second, a graphical representation of the distribution system in the form of a binary tree is created. This happens for both the original and the optimized system. However, the calculation of these binary trees and the time it takes to view them makes the timer inaccurate, so time

We were lucky in that we found code written by Hadi Saadat and distributed with his textbook, "Power System Analysis, 2nd Edition" that was very similar to what we needed for creating the initial Jacobian and solving the system of nonlinear equations using the Newton Raphson iterative method. However, since we also wanted to optimize the system, we had to modify (or completely rewrite) a lot of the code to incorporate the extra data, loops, and subroutines we were adding.

Flowchart of Newton-Raphson:



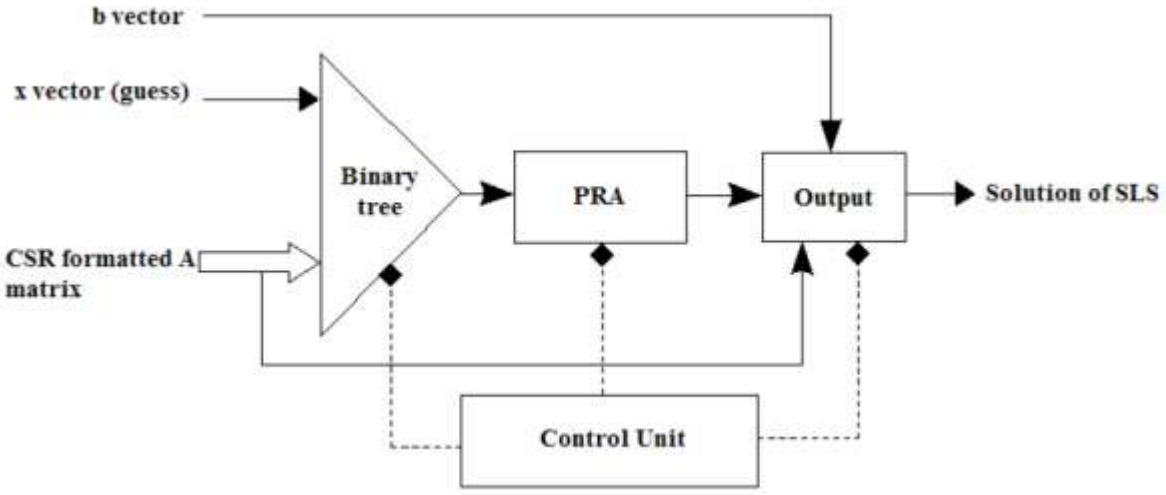
Newton-Raphson as performed on FPGA

Jacobi Method: Jacobi method is used for this project because it is more efficient and shows more parallelism.
It is an algorithm for determining the solutions of a system of linear equations with largest absolute values in each row and column dominated by the diagonal element.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

$$x_i^{(t+1)} = \frac{b_i - \sum_{j=1, j \neq i}^{n} a_{ij} x_j^{(t)}}{a_{ii}}$$

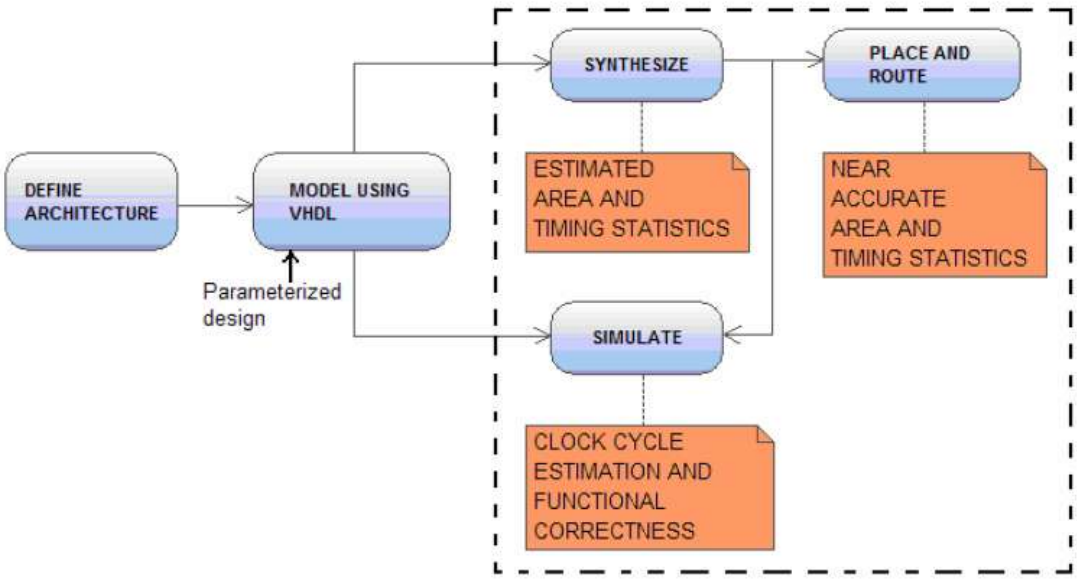
In the above equation Jacobi method $x_i^{(t+1)}$ is dependent on $x_j^{(t)}$ so the values of $x_j^{(t)}$ can be fed every clock cycle without stalling the pipeline.



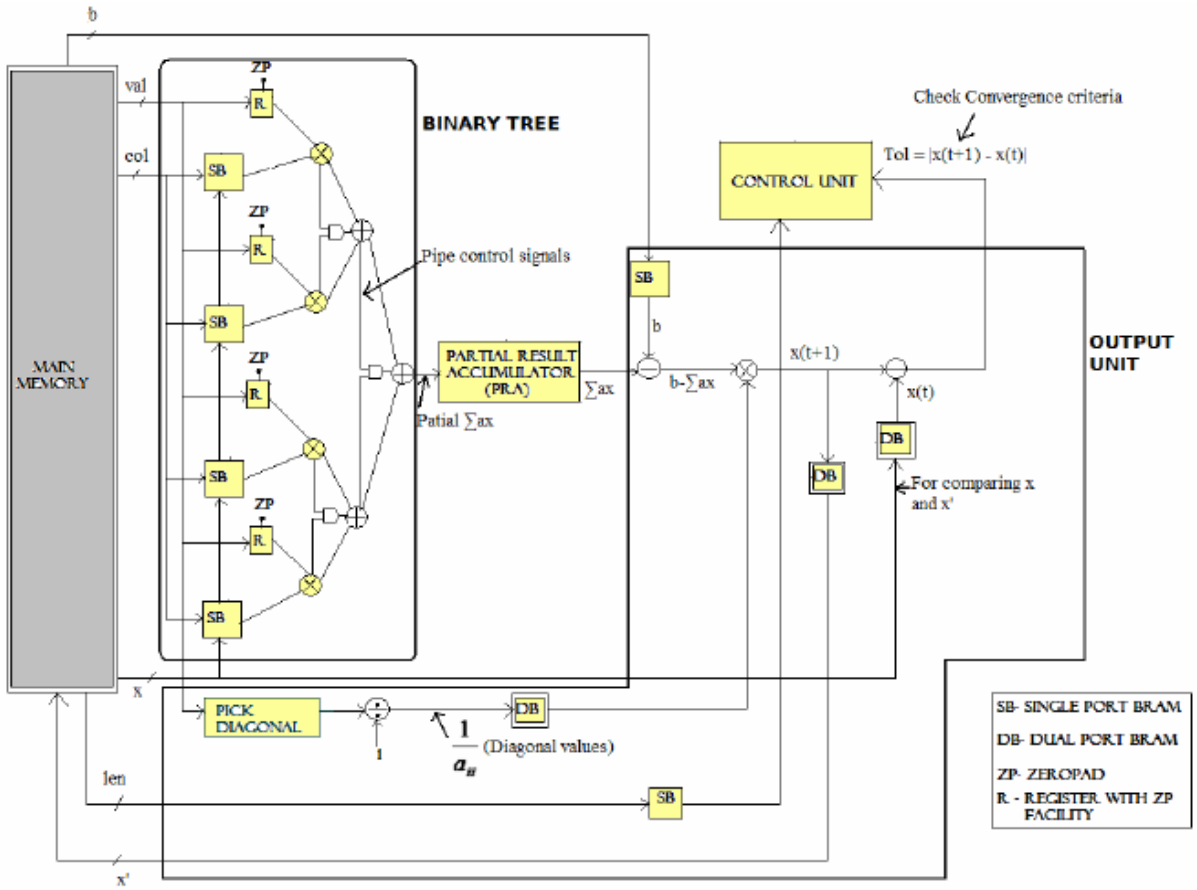
Block diagram of Sparse Matrix Jacobi Solver

Binary tree takes into its multiplier unit the packets of CSR row matrix obtained by applying the Compressed row matrix algorithm on Matrix A along with the 1 x n column X matrix. The multiplied result is then stored into Partial Result Accumulator (PRA) where it waits for the clear b vector signal. Once the b vector is supplied, the ∑ax is supplied to the control unit for the subtraction and division operation.

Hardware Design Flow:



Architecture of Sparse Matrix Jacobi Solver:



The execution starts with the initialization phase :

- X vector is loaded into each of the single port block RAMs in the binary tree unit and in the double port block RAMs in the output unit.
- b and len vectors are loaded in the single port block RAMs in the output unit.
- Packets of val(i) and col(i) are sequentially supplied to binary tree unit after the initialization phase.
- X vector is supplied to the multiplier unit after reading the col(i) vector input.
- As soon as control unit detects the input of A vector inputs, the zeropad signal of corresponding register R supplies a 0 to the multiplier unit.
- Val (i) is supplied to the divider unit where the divider unit calculates the reciprocal of vector A and stores it in double port block RAMs.
- The multiplied result ∑ax is stored in PRA and waits for control unit to input B vector.
- Both B vector and ∑ax is supplied to the Subtractor unit to calculate b - ∑ax.
- It is then multiplied with the reciprocal of A vector to obtain the result of next iteration $x_i^{(t+1)}$ which is stored in dual port block RAMs.

In the final step convergence condition is checked which is obtained by subtracting the input from output. If the condition is satisfied then there is no need for another iteration. If the condition is not satisfied then the pipeline is flushed and whole process is repeated again until the condition is satisfied.

Packet Method:

We use the packet method to supply the value of a and x so as to reduce the execution time used in calculating ∑ax and also to avoid stalling the pipeline.

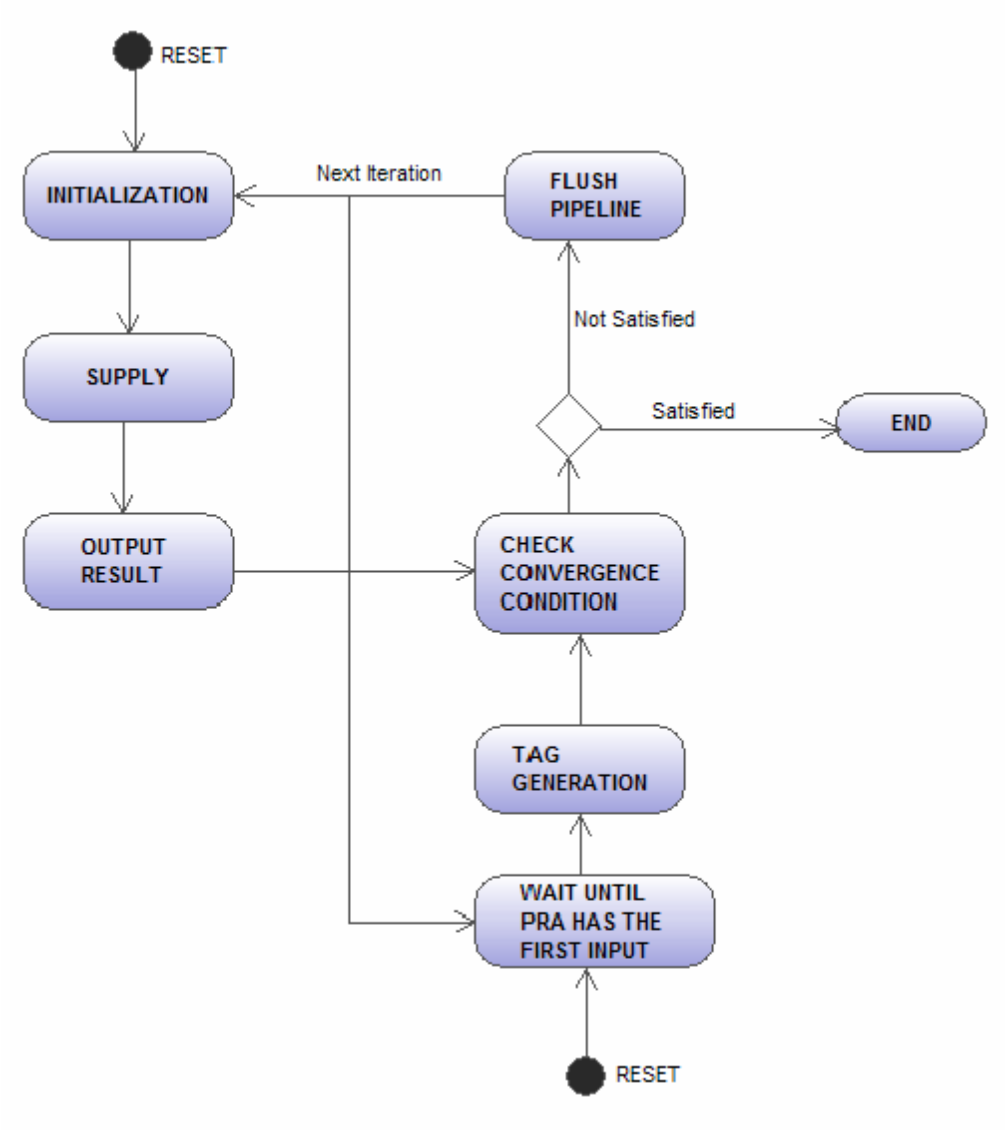
row	=	2 | 4 | 5 | 6 | -12 | 12 | -11
$packet_1$	=	2 | 4 | 5 | 6
$packet_2$	=	-12 | 12 | -11 | 0

Divide operation:

Since division is more time consuming than all other algebraic function like addition, subtraction and multiplication so instead of directly dividing our value we perform the operation of subtraction and multiplication. The division is performed in three basic steps:

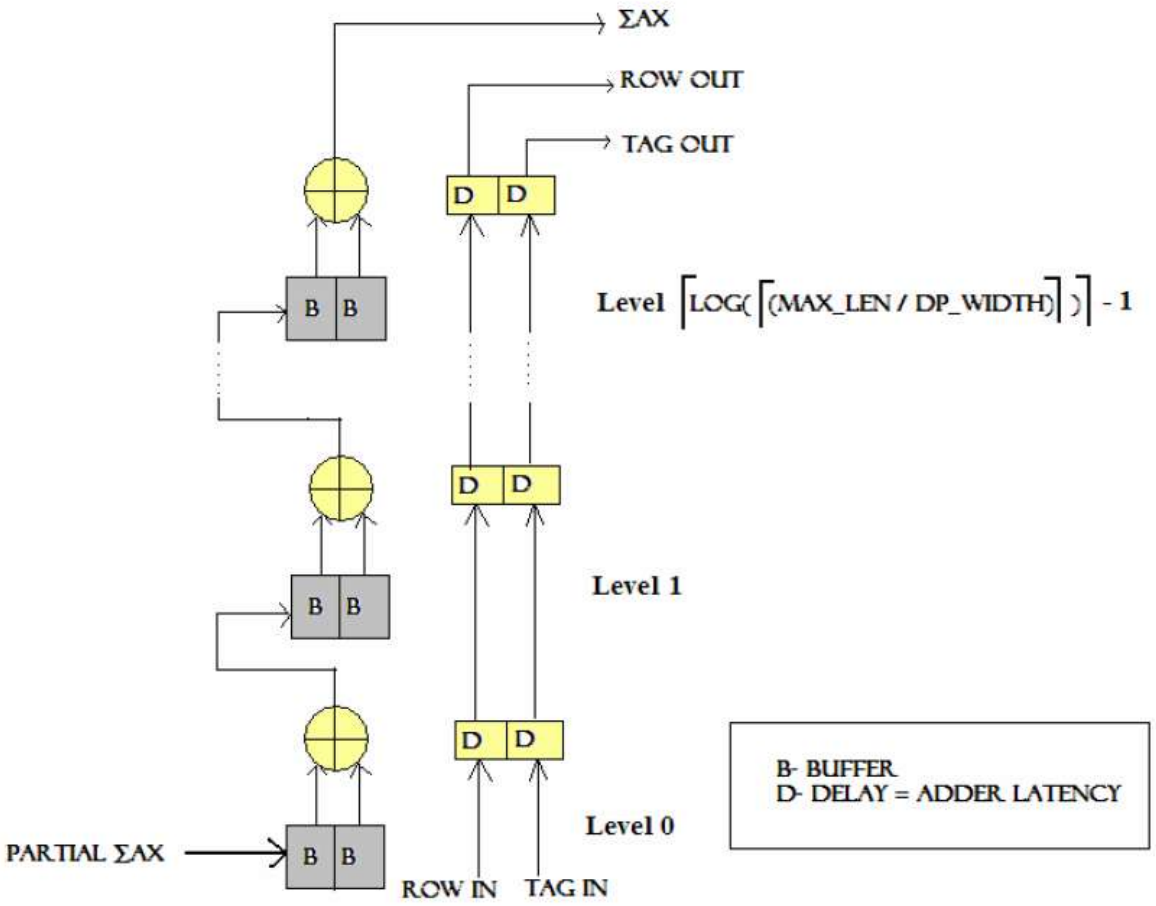
1) Calculate the reciprocal of *aii*
2) Calculate the numerator b - Σax
3) Multiply the results obtained from steps 1 and 2.

Sparse Matrix Jacobi Method:



a) **Initialization:** The x vector is stored in each of the single-port block RAMs connected to the multiplier unit of the binary tree.
b) **Supply:** Packets of Val and Col vector are supplied to the single port block RAM. Correspondingly Col vector is used to look up the value of x.
c) **Output Result:** Control Unit stores the output in the dual port block RAMs
d) **Convergence condition:** Convergence condition is satisfied when it is greater than the difference of input from output. If true there is no need for further iteration.
e) **Pipe Flushing:** If convergence is not satisfied pipeline is flushed and iterates again.

Partial Result Accumulator(PRA):



The Algorithm followed for PRA:

Buffer(B1)
supply the partial Σax to the buffer
If(tag >= 2 level_id +1) then
If(B2 waiting for input) then
supply the value to the buffer B2
Else
supply the value to the adder
Endif
Else
If(B2 waiting for input) then
Zeropad the buffer
Endif
Supply the value to the adder
Endif

Send done to indicate that the adder can start working
Buffer(B2)
If(zeropad is false) then
Supply the value to the adder
Else
Zeropad the buffer
endif

CSRConv Description:

In matrix equations zeros take up memory and slows down the computations that are taking place. The CSRConv program takes a matrix with several zeros in the matrix and converts it into five arrays with no zeros. The CSRConv programs scans the matrix for all the non zero elements and turns the matrix into the five following arrays:
Index: The number of nonzero elements in the matrix
Val: each individual non-zero element of the matrix
Col: what column the corresponding non-zero element is found in the matrix
rowindex: the index value of the first non-zero element in each row, the last element of this array is the total number of non-zero elements plus one
len: the value of the previous value of rowindex minus the current value of rowindex

Original Matrix

3	0	1	0	0
0	4	0	0	0
0	7	5	9	0
0	0	0	0	2
0	0	0	6	5

Compressed Sparse Row Format

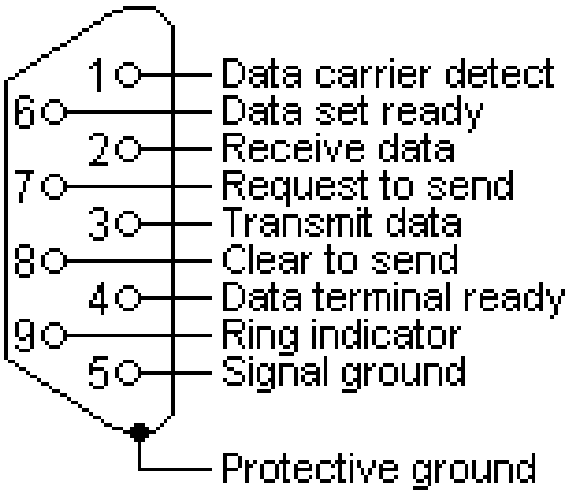
Index:	1	2	3	4	5	6	7	8	9
val	3	1	4	7	5	9	2	6	5
col	1	3	2	2	3	4	5	4	5
rowindex	1	3	4	7	8	10			
len	2	1	3	1	2				

Communication between MATLAB and the FPGA

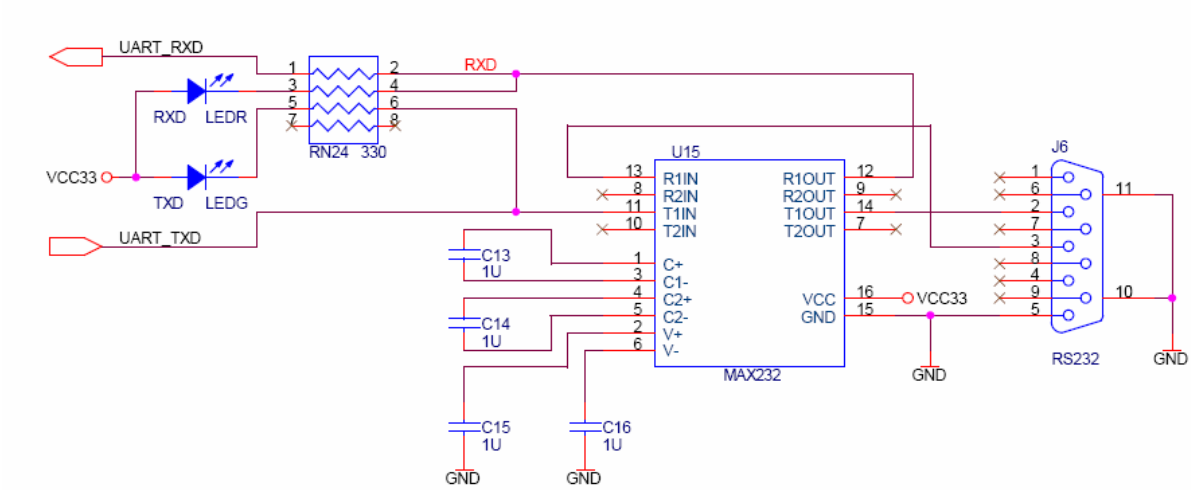
For our communication between the FPGA and MATLAB we had initially planned on using a VHDL function called TextIO, which was supposed to let us read and write file to and from the connected computer. However, it turned out that this function was only meant to be

used as a test bench function for simulating data input, and is not actually synthesizable on the actual FPGA. Once we figured this out, we turned our attention to serial communication.

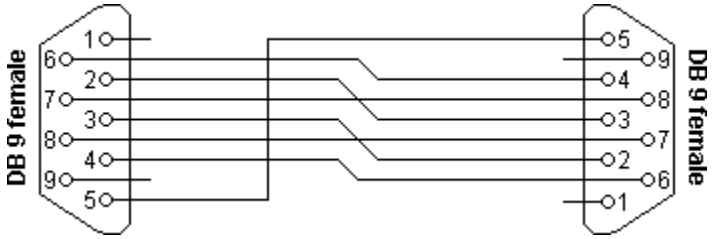
We had a few different options for serial communication with the FPGA development board we were using. They were Ethernet, USB, and RS-232. After doing some research, we determined that Ethernet was a very complex and touchy system to implement. We also found that USB was considered unreliable for mass data transfers, as some bits tended to get lost in the transmission. RS-232, however, seemed to offer what we needed. A pinout diagram of an RS-232 connector can be seen below.



In order to use the RS232 port, we needed to find a way to open the port both in MATLAB and on the FPGA itself. Doing this in MATLAB is fairly simple, and used the same general forms to open and close the serial port and transmit data as the code that we had previously written to open and close text files and read/write data to them. Doing it on the FPGA is a bit more complicated, and requires a specific UART (Universal Asyncronous Receiver/Transmitter) to work. The UART that came with our board was written in Verilog, which we could not use as the rest of our project has been done in VHDL. We were able to find some VHDL UART code online that we were able to use for our project. A schematic of the RS-232 circuit found on our Altera DE2 development board can be seen below.



We then soldered up a null modem cable (cable with crossed pins to communicate between to data terminals as opposed to a data terminal and a data receiver) so we could communicate via MATLAB from one computer to another. Using this, we were able to verify that the MATLAB code we had written to open and close the port as well as transmit and receive the data was working properly. A pinout of an RS-232 null modem cable can be seen below.



After adjusting the UART, flow control settings, and pin assignments on the FPGA, we were able to get communication to work. Unfortunately, we were not receiving the same data that we were sending. For example, we would try to send a 5, but would receive a 5.3. This was an issue that we did not resolve, as we discovered the RS-232 communication was slower than we had anticipated and would not result in the speedup we desired. It took over a second to send and receive data for just one iteration via RS-232, while our entire algorithm could be run in a fraction of the time on MATLAB.

Project Comments:

Problems Encountered and Troubleshooting:

During the testing and troubleshooting phases of writing our code we ran into a few problems, but none of them were very difficult to solve. Most of it was simple debugging.

The part of the project that we did have some troubles with was with our serial communication. Flow control was a problem, as we were not able to use either hardware or software handshaking. Hardware handshaking was not useable due to the fact that our FPGA Board only connected the send, receive, and ground pins, and hardware handshaking requires that you also be able to use the clear to send and request to send pins. We could not use software handshaking because it uses start and stop bits to exercise flow control, and the data we were

sending/receiving could have accidentally turned it on or off at the wrong time. Fortunately MATLAB does have the option of setting flow control to none, which is always ready to receive when the port is open.

Even once we got the flow control issues sorted out, we still had problems with our serial communication. We would send arrays of integers to the FPGA, but it would just receive ASCII values. After changing some pin assignments, we were able to get the FPGA to send values to MATLAB, but they would arrive distorted, as though there were some kind of noise or interference. For example, we would try to send a 5, but we would receive a 5.3 on the other end. Unfortunately we were unable to resolve this issue.

However, in our experiments with RS-232 serial communication, we discovered that sending data in this fashion was not nearly as fast as we had hoped. It took over a second to open the port, send a relatively small Jacobian matrix, and close the port. Our entire algorithm, including optimization, could be run in the strictly software environment more quickly than that. This led us to the conclusion that our goal of hardware acceleration was unrealizable using RS-232 communication due to these limitations with the serial communication and the MATLAB overhead associated with them.

Future Work:

There are several opportunities for future work on this project. The main area of future work would be to incorporate the FPGA chip and the processor on the same chip to eliminate the bottleneck of the communication between MATLAB and the FPGA. With the FPGA on the same board as the processor the communication could be performed faster than with the RS-232.

A model board can be developed to model a radial distribution system to take real world inputs and to run the optimization code. A low voltage board with circuit breakers, switches and relays could show how the project can optimize a radial system.

These are just a few of the options that could be done for future work, there are many more options that could be done.

Budget:

In order to implement this project from scratch you would need an Altera Development board and a MATLAB license which would cost about $750. Our Group was lent the development board and had MATLAB available to use through school computers, therefore we did not have to spend any of the allotted budget for the project.

Part	Quantity	Retail Cost	Expected Cost	Total Cost	Notes	Spent to date
Altera FPGA Development Board	2	$650	$0	$0	1 Board provided by Advisor	$0
Hadi Sadat Power System Analysis book	1	$160	$160	$160		$0
MATLAB License	2	$100	$100	$200		$0
Misc.	1		$72	$72		$0
	Total Retail	$910	Total Expected	$432	Total to Date	$0

Summary:

In summary, we were unable to achieve our overall goal of obtaining a speedup in the power flow solution using FPGA hardware acceleration due to limitations with our serial communication process. We were, however, able to meet most of our other requirements. We identified that the software bottleneck in the way the power flow solution is currently calculated was due to the additional time taken to solve the system of nonlinear equations involved, mostly due to the sparse jacobian matrix. Our software was able to run the Newton Raphson method successfully, both in MATLAB and VHDL. Our software was also able to optimize a radial distribution system using the Newton Raphson method. The only place we really ran into trouble was with our serial communication. Even after settling all of our issues with the UART, flow control, and various software problems we were still receiving some incorrect values. This became a moot point, however, when we realized that the serial communication was actually slowing down our algorithm.

We hope that this project is picked up in the future by another group, who can use our work as a stepping stone to making this into a real hardware accelerator. This would probably best be accomplished by incorporating the FPGA itself onto the same board as the processor, in order to cut out the serial communication that slowed down our design. We feel that working on this project has been a very interesting and educational experience, and hope to see future work in this area.

Appendix:

All of the following files can be found on the FPGASolve CD

Top level code can be found in full at the end of this document

Top level codes are in bold below

MATLAB Code:

- **MATLABtreealgo.m**
- CSRConv.m
- getdata.m
- getfamily.m
- getlineflow.m
- getybus.m
- linedatas.m
- **newtonraphson.m**
- serial.m

VHDL Code:

- async_reciever.v
- async_transmitter.v
- add_sub.vhd
- and_gate.vhd
- **binary_tree.vhd**
- **control.vhd**
- controlled_tree.vhd
- ctrl_red.vhd
- delay16bit.vhd
- delay16bit_N.vhd
- delay32bit.vhd
- multiplier2_12.vhd
- normalizer.vhd
- or_gate.vhd
- **package_sjs.vhd**
- supply.vhd
- swap.vhd
- tb_ctrl_red.vhd
- tree.vhd